RVC-CAL DATAFLOW IMPLEMENTATIONS OF MPEG AVC/H.264 CABAC DECODING

Endri Bezati[1], *Marco Mattavelli*[1], *Mickaël Raulet*[2]

[1] Ecole Polytechnique Fédérale de Lausanne, CH-1015 Lausanne, Switzerland
{firstname.lastname}@epfl.ch
[2]IETR/INSA Rennes F-35708, Rennes, France
mickael.raulet@insa-rennes.fr

ABSTRACT

This paper describes the implementation of the MPEG AVC CABAC entropy decoder using the RVC-CAL dataflow programming language. CABAC is the Context based Adaptive Binary Arithmetic Coding entropy decoder that is used by the MPEG AVC/H.264 main and high profile video standard. CABAC algorithm provides increased compression efficiency, however presents a higher complexity compared to other entropy coding algorithms. This implementation of the CABAC entropy decoder using RVC-CAL proofs that complex algorithms can be implemented using a high level design language. This paper analyzes in detail two possible methods of implementing the CABAC entropy decoder in the dataflow paradigm.

1. INTRODUCTION

1.1. RVC Standard

The purpose of the MPEG RVC standard is to offer a more flexible use and faster path to innovation of MPEG standards in a way that is competitive in the current dynamic and evolutive environment. This is meant to give MPEG standards an edge over its competitors by substantially reducing the time for which technology is developed and the time the standard is available for market applications. The RVC initiative is based on the concept of reusing commonalities among different MPEG standards and provide possible extensions by using appropriate higher level specification formalisms. Thus the objective of the RVC standard is to describe current and future codecs in a way that makes such commonalities explicit, reducing the implementation burden by providing a specification that is starting point closer to the final implementation. So as to achieve this objective, RVC provides the specification of new codecs by composing existing components and possibly new coding tools described in modular form [1, 5].

This work is part of the ACTORS European Project (Adaptivity and Control of Resources in Embedded Systems), funded in part by the European Unions Seventh Framework Programme. Grant agreement no 216586

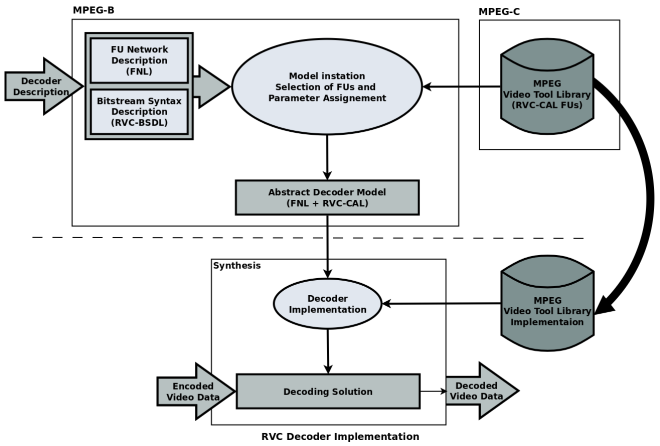

Fig. 1. Components of the MPEG RVC Framework.

The MPEG-B standard defines the languages that are used to build the MPEG RVC framework. The RVC-CAL dataflow programming language is the core of the system since it the language describing the behavior of each module called in RVC a Functional Unit (FU). With the specification of the FU network topology the functional behavior of a video decoder, called also abstract decoder module is fully specified. The term abstract refers to the fact that FUs are only characterized by the I/O behavior and by the firing rules embedded in the RVC-Cal language. Thus the interaction of each FU with interconnected Fus is fully specified by abstracting time and by only defining dependencies of dtat generation and consuption. The MPEG-C standard defines the library of the video coding tools (Video Tool Library or VTL). Figure 1 illustrates the concept that from any abstract decoder model constituted by a Functional Unit network description decoding solutions can be derived for hardware or software implementation.

1.2. CAL Language

CAL Actor Language is a language based on the Actor model of computation for data flow systems. It provides many natural concepts to facilitate modeling of those systems. An actor, or equivalently an FU, is a modular component that encapsulates its own state. Each actor interacts with each other through FIFO channels, see Figure 2. An actor in general may contains state variables, global parameters, actions, procedures, functions and finite state machine that controls the executions of actions. CAL enables concurrent development and provides strong encapsulation properties. CAL is used in a variety of applications and has been compiled to hardware and software implementations. The RVC-CAL language is a subset of the CAL language and it is normalized by ISO/IEC as a part of the RVC standard. Although it has some restrictions in data types and features that are in used in CAL [1, 3] is sufficient and efficient for specifying streaming and signal processing systems such as MPEG compression technology.

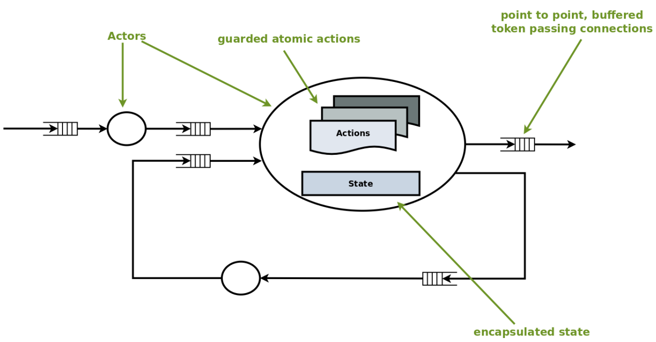

Fig. 2. The CAL computing model

1.3. Open RVC-CAL Compiler

The Open RVC-CAL Compiler is a tool set which provides developers with a compiler infrastructure able to generated source code in several imperative languages starting from a network of RVC-CAL actors and the XDF network topology description [8]

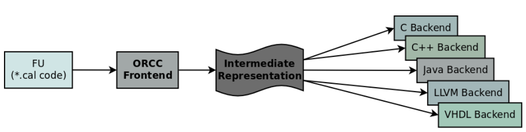

Fig. 3. ORCC Framework

- **Functional Unit (FU):** is a video processing component or an actor in CAL language. The RVC framework provides a Video Tool Library which consists a set of FUs that can be embedded by the client and combined for building the required decoder
- **Frontend & IR :** For a RVC-CAL dataflow program to be compiled, its compilation process is done in a two-step process. The fronted will parse all actors and it will translates them to an Intermediate Representation (IR) which is serialized to one file per actor in a JSON-based format (JavaScript Object Notation). The IR is a data structure that is constructed from input data (the actor) to a program, and from which part or all of the output data of the program is constructed in turn. The next steps is to run a language target-specific back-end.
- **Backends:** Depending on the target, ORCC offers a variety of back-ends. Their purpose is to create target specific code. Each backend will parse the hierarchical network from a top-level network and its child network. Also optionally it flattens the hierarchical network. ORCC for the moment offers a variety of back-ends. These back-ends are C, C++, LLVM, VHDL and a partial support for the Xlim code generation.

To generate a software decoding solution we used the C backend of ORCC. The generated C code is ANSI-C compatible and it is portable to different platforms such as Windows, Linux, Mac OS X and others.

1.4. Entropy Encoding

In information theory an entropy encoding is a lossless data compression scheme that is independent of the specific characteristics of the medium. In the AVC/H.264 three types of entropy encoding are used. Exp-Golomb for decoding the headers of the bitstream (video file). Furthermore the description of a macroblock and its quantized coefficients are encoded using the CAVLC or CABAC encoding.

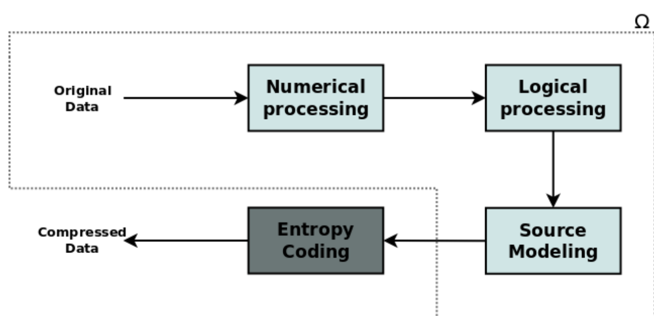

Fig. 4. Entropy encoding as the final stage of compression

2. CABAC ENTROPY DECODER

CABAC or Context Adaptive Binary Arithmetic Coding is an entropy decoder that provides higher compression rates than the tools available in the Baseline Profile such as the CAVLC (Context Adaptive Variable Length Coding) entropy decoder. CABAC selects a probability model for each syntax element, it adapts the estimated probability based on local statistics and it uses an arithmetic coder rather than the classical variable-length coding (CAVLC). The CAVLC is a certainly easier to implement and require less processing resources, it gets the value directly by reading the bit-stream. Conversely CABAC presents a higher implementation complexity and requires a feedback loop between the context modeler and the arithmetic coder [2].

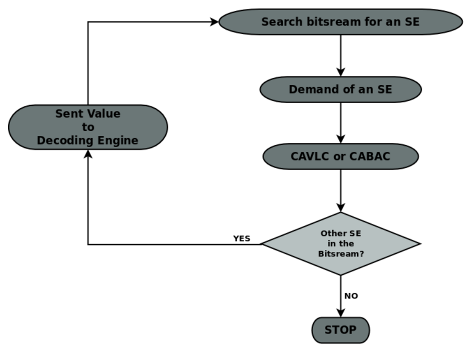

Fig. 5. An simple interpretation of the H.264/AVC parser.

Depending on the context, the AVC/H.264 parser demands to the entropy decoding processing engine to decode the value of a Syntax Element (SE), see Figure 5. A SE is an element data represented in the bit-stream, for instance a Macro-block type(Intra4x4, Intra16x16 or others).

2.1. Binarization Process

Each SE has a particular binarization. In H.264/AVC four types of binarization are defined.

- Unary binarization, used for reference picture list re-ordering SE
- Truncate unary binarization, used for the Intra chroma prediction mode SE
- Concatenated unary/k-th order Exp-Golomb(UEKg) binarization, used mainly in the Residual block process and in the movement prediction for P and B slices
- Fixed Length binarization, used for flag types SE

An example of the Fixed Length binarization process can be seen in the table below. The binIdx, indicates the binary index of the binary string. For example this same binary strings indicates the values of the rem_intra_4x4_pred_mode and rem_intra_8x8_pred_mode SE from the H.264/AVC standard.

2.2. CABAC Variables

In the H.264/AVC standard the prediction of the binary value is basically calculated by these variables.

- ctxIdx, which indicates the context of the syntax element

Value of Syntax Element	Bin String
0	0 0 0
1	1 0 0
2	0 1 0
3	1 1 0
4	0 0 1
5	1 0 1
6	0 1 1
7	1 1 1
binIdx	0 1 2

Table 1. FL binarization with cMax = 7.

- codIRange and codIOffset, corresponds to the status of the arithmetic decoding process
- pStateIdx, corresponds to the probability state index
- valMPS, corresponds to the value of the most probable symbol

2.3. CABAC Arithmetic coding engine

After the binarization of an SE, each bin in the binarized string will be given a context (ctxIdx) and then encoded using the CABAC arithmetic coding. The Context Arithmetic coding theory is based on the principle of recursive interval subdivision. Given a probability estimation p_0 and $p_1 = 1 - p_0$ of the binary decision (0,1), codIRange will be initially gives the code sub-interval, this range will be subdivided into two sub-intervals having the following range $p_0 * codIRange$ and $codIRange - p_0 * codIRange$, respectively. Depending on the decision, the corresponding sub-interval is going to be chosen as the new code interval, and the binary code string pointing into that interval will represent the sequence of observed binary decisions. In the decoding decision procedure it is useful to distinguish between the most probable symbol (MPS) and the least probable symbol (LPS), it is better that the binary decisions have to be identified as either MPS or LPS, rather than 0 or 1. Given this terminology, each context is specified by the probability p of the LPS and the value of MPS (valMPS), which is either 0 or 1. [7]

3. RVC-CAL CABAC IMPLEMENTATION

The implementation of the RVC H.264/AVC parser FU in the current RVC-CAL AVC CBP (Constrained Base Profile) [4], [6] decoder is being used as a base for the CABAC entropy decoder. That means that each action used by the parser from the CAVLC is modified so that can support CABAC SE's extraction. The CABAC implementation is modeled by twelve untagged actions. These actions are not named in the source code because they are executed outside the FSM (Finite State Machine) of the parser FU. Here we give them names for clarity. We developed two methods to retrieve the Syntax Elements values from the AVC/H.264 parser Functional Unit. The first method is using a Look Up Table (LUT) for comparing the decode binary string given by the CABAC procedure and the binarized Value of an Syntax Element. This method was later abandoned due to the complexity of the binarization LUT of the quantized coefficients (Macroblock residual values) and its slow double loop algorithm. Given the difficulty of the binarized residual values, we developed a new method, Bin By Bin, which is an adaptive decoding solution for each Syntax Element.

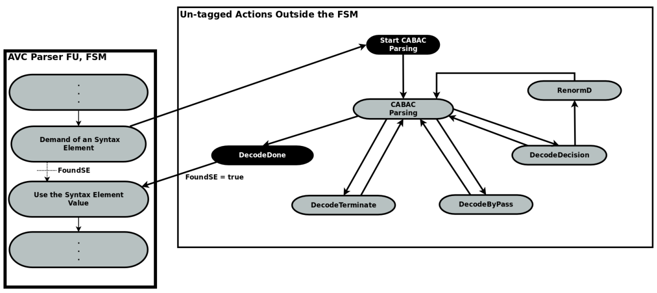

Fig. 6. RVC-CAL main CABAC parsing actions

In the Figure 6 we represent the LUT and Bin by Bin method as CabacPasrsing, depending the method we want to use. CabacPasrsing is the main action and it controls the activation of the CABAC decoding actions. Depending the SE three actions are used to decode the binary value. These actions are DecodeDecision, DecodeByPass and DecodeTerminate. All three of them execute smaller actions to read the bitstream.

3.1. Look Up Table method

After the CABAC arithmetic decoding the decoded bins have to be de-binarized so the value can be retrieved. A very simple way to get the decoding value from CABAC de-binarization is to use a Look Up Table. As each SE has a specific binarization the idea was to create a look up-table for each SE and to stock the binarized values in a multi dimension array. Thus index of the array will be the decoded value and the rows of the table represents binarized bin string. With this method bits are recovered one by one and then are compared with the binarization table of the searched SE. As we mentioned before we abandoned this method, but all the SEs expect the coeff_abs_level_minus1 (quantized coefficients), ref_lx and mvd_lx (mouvement vectors) fully functional.

3.2. Bin by Bin method

Using a LUT demands more memory and the comparison algorithm could be much more time consuming. For example to decode the 255 value of the coeff_abs_level_minus1 SE with LUT method, the algorithm needed to execute 255*(n

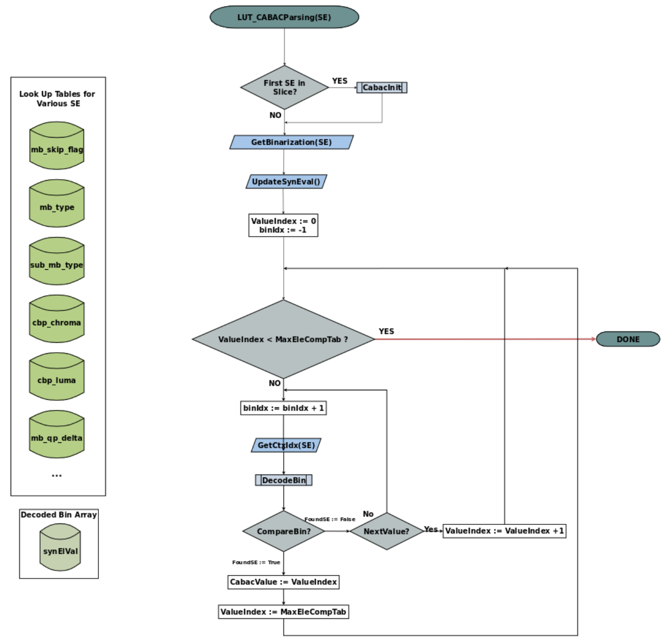

Fig. 7. The LUT Method

bins) double loops. So to make the process faster we developed the Bin By Bin method. In this method each SE has a different de-binarization procedure. In this way the CABAC decoding is unique for each SE.

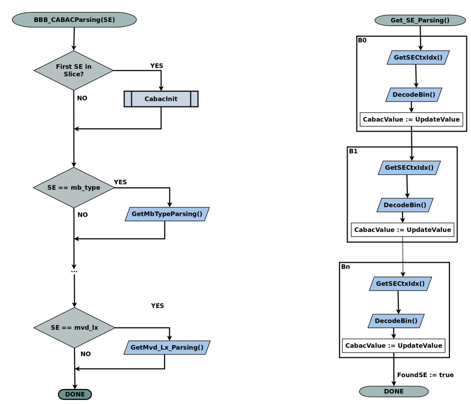

Fig. 8. The Bin By Bin Method

The Figure 9 shows a part of the coeff_abs_level_minus1 SE de-binarization process, this procedure is called as many times as the SE value is completely de-binarized.

4. RESULTS

4.1. Conformance sequences support

The RVC-CAL CABAC implementation supports all the SEs found in the AVC/H.264 standard except the mb_field_decoding_flag (which indicates if the macroblock

```
procedure CoeffAbsParsing()
begin
  if StartPrefix then
    if binValDecoded = false then
      binIdx := binIdx + 1;
      //give the ctxIdx
      ctxIdxCoeffAbsLevel();
      //Get the bin Value
      StartDecodeDecision := true;
      CabacStartProcess := false;
      NextBitToDecode := false;
      binIdx := 0;
    end
    ...
  if binValDecoded = true then
      // Increment binIdx
      binIdx := binIdx + 1;
      if binVal = 1 then
        CabacValue := -1;
      else
        CabacValue := 1;
      end
      //Finishing the Prefix for a 0 binVal
      StartPrefix := false;
      numDecodAbsLevelEq1 := numDecodAbsLevelEq1 + 1;
      //Stop cabac process and give the value
      CabacStartProcess := false;
      FoundSE := true;
  end
  ....
```

Fig. 9. A part of the coeff_abs_level_minus1 SE debinarization procedure.

is field coded). Also because the abstract decoder model used in this implementation is coming for the AVC CBP FU, it does not support the decoding of the BiPred slices. Thus our CABAC implementation supports all the encoded Macroblock SE except those coded in field order. The following Table 2 is a list of supported conformance AVC/H.264 sequences coded using the CABAC entropy encoding. Because of the AVC CBP FU the sequence marked as *IPB* are supported only by the AVC Parser FU, the decoding of those sequences is a work in progress of the AVC FRExt (Fidelity Range Extensions) decoder FU.

4.2. RVC-CAL CABAC versus JM CABAC implementation

The RVC-CAL is by design intended to be a high level language. The Decode Decision process is used by almost all Syntax Element in the H.264/AVC standard. Here we compare the interpretation of the H.264/AVC Decode Decision flowchart(Figure 10) with the RVC-CAL (Figure 11) and the C language used in the JM software Reference software of the H.264/AVC Standard (Figure 12).

In term of source code we have approximately the same number of code lines for this two different implementations, but the difference is in the readability and the understanding of the source code. The RVC-CAL implementation results to be an identical copy the of the Decode Decision flowchart. In addition the C code of the JM software contains pointers, the variables are not named exactly the same and in general the

Name	Slice Type	N. Frames	Status
CABA1_Sony_D	I	50	Passed
CABA2_Sony_E	IP	300	Passed
CABA3_Sony_C	IPB	300	Only Parser
CABA3_TOSHIBA_E	IP	300	Passed
CABA1_SVA_B	I	17	Passed
CABA2_SVA_B	IP	17	Passed
CABA3_SVA_C	IPB	33	Only Parser
CANL1_TOSHIBA_G	I	300	Passed
CANL1_Sony_E	I	17	Passed
CANL2_Sony_E	IP	300	Passed
CANL3_SVA_C	IPB	300	Only Parser
CANL1_SVA_B	I	17	Passed
CANL2_SVA_B	I	17	Passed
CANL3_SVA_B	IP	17	Passed
CANL4_SVA_C	IPB	33	Only Parser
CAQP1_Sony_B	IP	50	Passed
CACQP3_Sony_D	IPB	50	Only Parser

Table 2. Conformance sequences supported by the Bin By Bin RVC-CAL implementation

code readability is not as easy to read and analysable as the RVC-CAL code.

Our full implementation of the CABAC in RVC-CAL contains less source code than the C implementation of the JM AVC decoder. 2500 source code lines for the RVC-CAL and more than 3000 lines for the C code.

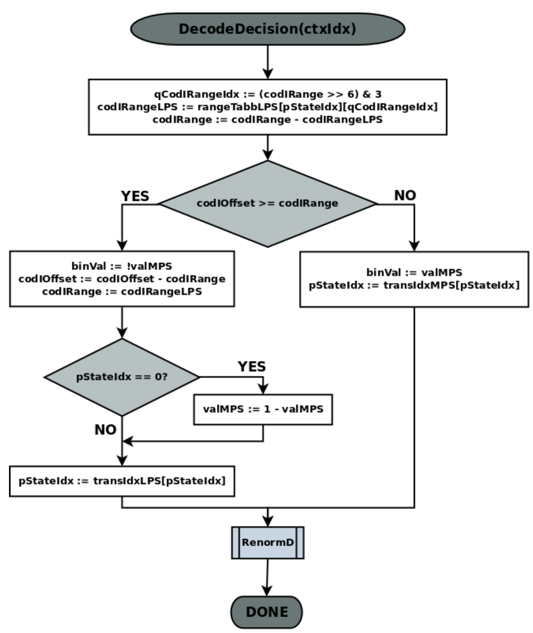

Fig. 10. Decoding Decision flowchart in the H.264/AVC standard.

5. TO A DATA FLOW MODEL

The actual form of the AVC parser in RVC-CAL is sequential, as supposed to be for a parser. As RVC-CAL is a data

```
DecodeDecision: action ==>
guard
  StartDecodeDecision and
  bypassFlag = false and ctxIdx != 276 and
  StartRenormD = false
do
  qCodIRangeIdx := ( codIRange >> 6 ) & 3;
  codIRangeLPS :=
    rangeTabLPS[pStateIdx[ctxIdx]][qCodIRangeIdx];
  codIRange := codIRange - codIRangeLPS;

  if codIOffset >= codIRange then
    binVal      := intXOR(valMPS[ctxIdx]);
    codIOffset  := codIOffset - codIRange;
    codIRange   := codIRangeLPS;

    if pStateIdx[ctxIdx] = 0 then
      valMPS[ctxIdx] := 1 - valMPS[ctxIdx];
    end
    pStateIdx[ctxIdx] :=
      transIdxLPS[pStateIdx[ctxIdx]];

  else
    binVal := valMPS[ctxIdx];
    pStateIdx[ctxIdx] :=
      transIdxMPS[ pStateIdx[ctxIdx] ];
  end

  // Give the right to compare in CabacParsing
  binValDecoded := true;
  // write the bit into the  ReadBinString
  ReadBinString[binIdx + 1] := binVal;

  if codIRange < 0x0100 then
    StartRenormDRead := true;
  else
    CabacStartProcess := true;
  end
  // Do not call again DecodeDecision
  StartDecodeDecision := false;
  // Do not give the hand to CabacParsing
end
```

Fig. 11. Decoding Decision action written in RVC-CAL.

```
unsigned int biari_decode_symbol
                 (DecodingEnvironmentPtr dep,
                  BiContextTypePtr bi_ct )
{
  unsigned int bit    = bi_ct->MPS;
  unsigned int *value = &dep->Dvalue;
  unsigned int *range = &dep->Drange;
  uint16       *state = &bi_ct->state;
  unsigned int rLPS   =
    rLPS_table_64x4[*state][(*range>>6) & 0x03];
  int *DbitsLeft = &dep->DbitsLeft;

  *range -= rLPS;
  if(*value < (*range << *DbitsLeft))   { //MPS
    *state = AC_next_state_MPS_64[*state]; // next state

    if( *range >= QUARTER ){
      return (bit);
    } else
      *range <<= 1;
    (*DbitsLeft)--;
  } else { //LPS

    int renorm = renorm_table_32[(rLPS>>3) & 0x1F];
    *value -= (*range << dep->DbitsLeft);
    *range = (rLPS << renorm);
    (*DbitsLeft) -= renorm;

    bit ^= 0x01;
    if (!(*state))
      // switch meaning of MPS if necessary
      bi_ct->MPS ^= 0x01;

    *state = AC_next_state_LPS_64[*state]; // next state
  }

  if( *DbitsLeft > 0 ){
    return (bit);
  } else {
    *value <<= 16;
    *value |=  getword(dep);
    (*DbitsLeft) += 16;
    return (bit);
  }
}
```

Fig. 12. Decoding Decision action written in C from the JM.

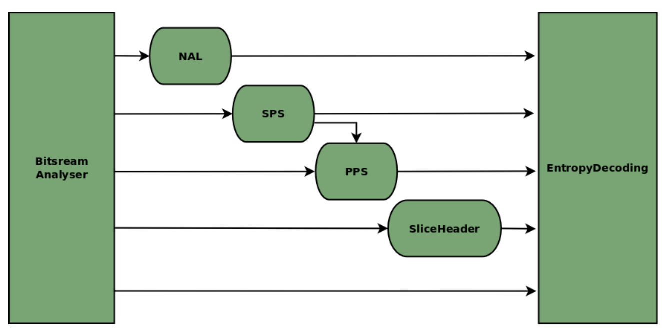

Fig. 13. A DataFlow model for the H.264/AVC Parser

flow based language, the idea here is to transform its sequential form to a data flow one. Actually the challenge is how to pipeline a sequential parser that includes feedback loops. As a matter of fact the bitsream has a start up code when a new slice starts. We could store the whole bitstream for a slice to buffer and then analyzed it and dispatched it to where is necessary. Actually in this way we could partition the parser process in smaller actors and create a pipeline architecture. These actors are the BitstreamAnalyser, NAL, SPS, PPS, SliceHeader and EntropyDecoding. Appart EntropyDecoding all other actors are explicit, for a reconfigurable decoder is a necessity to separate the CAVLC and the CABAC entropy Decoding. So in the EntropyDecoding actor we will regroup SliceData, macroblock_layer, mb_pred, sub_mb_pred, residual and residual_block [7]. The Figure 13 shows how the sequential AVC/H.264 parser can be transformed to a dataflow model using RVC-CAL.

6. CONCLUSIONS

Implementing the basic CABAC process in RVC-CAL took one month and two months for making it possible to decode the I,P and B slices. Our implementation is supporting all the CABAC encoded Syntax Elements found in the AVC/H.264 standard except the Macroblock field flag. The software generated decoder by our RVC-CAL abstract description can successfully decode a large part of the AVC/H.264 conformance sequences. Using RVC-CAL to interpret the H.264/AVC standard is really easy and this effectively reduces the conception time and actual work compared to other languages. With RVC-CAL is not only possible to implement a type of algorithm in one way, but it can be implemented as sequential monolithic process, as a monolithic dataflow model and as a networked data flow model. Thus implementing the AVC/H.264 Parser in a data flow model and its entropy decoding engines (CAVLC and CABAC) could achieve better performances than the current monolithic data flow model.

7. REFERENCES

[1] S. S. Bhattacharyya, J. Eker, J. W. Janneck, C. Lucarz, M. Mattavelli, and M. Raulet. Overview of the MPEG Reconfigurable Video Coding Framework. *Springer journal of Signal Processing Systems. Special Issue on Reconfigurable Video Coding*, 2009.

[2] H. Eeckhaut, M. Christiaens, D. Stroobandt, and V. Nollet. Optimizing the critical loop in the H.264/AVC CABAC decoder. *Field Programmable Technology, 2006. FPT 2006. IEEE International Conference*, 2006.

[3] J. Eker and J. Janneck. CAL Language Report. Technical Report ERL Technical Memo UCB/ERL M03/48, University of California at Berkeley, Dec. 2003.

[4] J. Gorin, M. Raulet, Y. Cheng, H. Lin, N. Siret, K. Sugimoto, and G. Lee. An RVC Dataflow Description of the AVC Constrained Baseline Profile Decoder. In *Proceedings of ICIP'09*, Nov. 2009.

[5] ISO/IEC FDIS 23001-4. *MPEG systems technologies – Part 4: Codec Configuration Representation*, 2009.

[6] J. W. Janneck, M. Mattavelli, M. Raulet, and M. Wipliez. Reconfigurable video coding: a stream programming approach to the specification of new video coding standards. In *MMSys '10: Proceedings of the first annual ACM SIGMM conference on Multimedia systems*, pages 223–234, New York, NY, USA, 2010. ACM.

[7] I. JTC1/SC29/WG11 and I.-T. S. Q.6. *Joint Draft ITU-T Rec. H.264 — ISO/IEC 14496-10 / Amd.3 Scalable video coding* , 2007.

[8] M. Wipliez, G. Roquier, and J. Nezan. Software Code Generation for the RVC-CAL Language. *Journal of Signal Processing Systems*, 2009.